CMPSC 462: Data Structures and Algorithms (Spring 2024)

Project-2: Search And Sort

Lily Stauffer

Computer Science

Instructor

Dr Vinayak Elangovan, Ph.D.

Program of Computer Science

Submitted on: October 14, 2024

Contents

[Introduction 3](#_Toc179822937)

[Background 3](#_Toc179822938)

[Searching Algorithms 3](#_Toc179822939)

[Sample Outputs 5](#_Toc179822940)

[Sorting Algorithm Output 5](#_Toc179822941)

[Searching Algorithm Output 5](#_Toc179822942)

[Time Complexity Analysis 6](#_Toc179822943)

[50,000 Random Integer Timetable 6](#_Toc179822944)

[100,000 Random Integer Timetable 6](#_Toc179822945)

[Conclusion 6](#_Toc179822946)

[References 6](#_Toc179822947)

[Appendix 7](#_Toc179822948)

[isSorted function 7](#_Toc179822949)

[Linear Search function 7](#_Toc179822950)

[Binary Search function 7](#_Toc179822951)

[Min search function 8](#_Toc179822952)

[Max search function 8](#_Toc179822953)

[Distinct search function 8](#_Toc179822954)

[Selection sort function 9](#_Toc179822955)

[Insertion sort function 9](#_Toc179822956)

[Bubble sort function 10](#_Toc179822957)

[Merge sort function 11](#_Toc179822958)

[Test Driver 12](#_Toc179822959)

## Introduction

A common problem within computer science and the general world is having to sort a set of things or having to search for a thing within a set of other things. How does one solve these problems efficiently? One might go through the list element by element and compare, but that’s not too efficient. Naturally, computer scientists came up with several ways to go about solving this problem.

There is a way to compare how fast algorithms are, using something called Big O notation. Generally Big O refers to only the leading term of the polynomial time complexity of an algorithm. It is sometimes called the order of growth, because it affects the time that an algorithm would take the most of any term.

## Background

There are two kinds of algorithms that will be discussed in this paper, searching and sorting. They both solve different problems, so they are both distinct.

### Searching Algorithms

#### Linear Search

This algorithm is probably the easiest to understand. You look at each element in a list, one by one, beginning at the start, and check to see if it is your target, the thing you are searching for. If it is, return true. If it isn’t, move on to the next element and repeat until there are no more elements. If there are no more elements, return false.

This has an order of growth of O(n), because it only goes through each element once.

#### Binary Search

This algorithm is less easy to understand when compared to Linear search, but it makes up for that in its efficiency. An important thing to note is that in order for this algorithm to work, the list needs to be sorted first. The way this algorithm works is it first goes to the middle of the list, and compares that value to our target, if that value is greater than our target, we know that our target must be to the left of that value, so we can effectively ignore everything to the right. Similarly, if the value is less than our target, we can ignore everything to the left. We repeat this process of dividing the list in half by comparing the element in the middle to our target, until we either find our target, or we run out of elements.

This has an order of growth of O(logn), because we divide the number of cases we have to look at in half each time.

#### Min/Max Search

These are grouped together because only one thing changes between them. The way that these algorithms work is by going through the list one by one and taking a running tally of whatever number is the greatest or the least. If it is greater than our greatest value, we swap out the greatest. If it is less than our least value, we swap out the least. We repeat this process until we get to the very end, then we return our value.

These both have an order of growth of O(n), because we go through our list once.

#### Distinct Search

This algorithm goes through our list, element by element, and checks to see if the element is in a set of all the elements we already have. If it is present already, it adds it to a list which will be returned at the end. If it isn’t present already, add it to the set and then move on to the next element. Interestingly the in operator on a set is constant speed, O(1), so it doesn’t impact the computation time when it comes to bigger and bigger lists.

The order of growth of this function is O(n), because we go through our list once.

### Sorting Algorithms

#### Selection Sort

This algorithm begins by skipping our first element. It does this because it attempts to create a sorted section at the start of the list. It does this by going through the list and finding the smallest unsorted element, and then it swaps and adds that to our sorted list at the beginning. It builds the sorted list up element by element.

The order of growth of this function is O(n^2), because we go through the list once for each element, we process n things, n times, giving us n^2.

#### Insertion Sort

This algorithm is like selection sort in that it attempts to create a sorted section at the beginning of the list. It does this by swapping elements into our list, it doesn’t need to be the next element in order though, unlike selection sort. This algorithm starts with the second element, then compares it to the first element, and swaps if needed. Then it moves onto the next element and goes through our sorted list at the beginning and swaps it into place if needed. Repeat until there are no more elements needed to be sorted.

The order of growth of this function is O(n^2), because we go through the list for each element, and then we go through it again for each element.

#### Bubble Sort

Bubble sort starts out by comparing the first two elements and sorting them. Then it looks at the next element and compares it to the latter half of the first two elements and swaps if needed. Then it moves on and compares it with the next element, and the next element, all until you get to the final element. This is sort of like reversed selection sort, where instead of creating a sorted list at the beginning, you create it at the end. And this loops through however many times is needed, because it checks to see if there is a swap made, if a swap was made, it repeats again. If no swap was made, the algorithm ends, because the list is sorted.

This algorithm has an order of growth of O(n^2) because it loops through the list of n elements n times.

#### Merge Sort

Merge sort starts out by dividing the list in two, and then dividing those lists in two, repeating until there are two elements. It sorts those two elements, then starts to merge all of those lists together.

This algorithm has an order of growth of O(n log n) because it merges log n lists in n time.

## Sample Outputs

### Sorting Algorithm Output

A screenshot of a computer program

Description automatically generated

### Searching Algorithm Output

A black screen with white text

Description automatically generated

## Time Complexity Analysis

### 50,000 Random Integer Timetable

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Linear | Binary | Min | Max | Distinct | Selection | Insertion | Bubble | Merge |
| 0.0029 | 0.00 | 0.0019 | 0.0009 | 0.632 | 69.54 | 75.185 | 177.96 | 0.181 |
| 0.001 | 0.00 | 0.0020 | 0.0009 | 0.020 | 66.546 | 83.805 | 157.14 | 0.174 |
| 0.001 | 0.00 | 0.003 | 0.0029 | 0.049 | 60.707 | 71.231 | 153.61 | 0.176 |

### 100,000 Random Integer Timetable

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Linear | Binary | Min | Max | Distinct | Selection | Insertion | Bubble | Merge |
| 0.0019 | 0.00 | 0.0019 | 0.0039 | 0.024 | 300.233 | 373.04 | 713.18 | 0.344 |
| 0.004 | 0.00 | 0.0029 | 0.0019 | 0.029 | 301.553 | 361.12 | 714.09 | 0.358 |
| 0.0019 | 0.00 | 0.0040 | 0.0029 | 0.020 | 275.262 | 332.81 | 690.54 | 0.382 |

As we can see from the two tables above, not all O(n^2) algorithms are created equal. Bubble sort takes the longest consistently, followed by insertion, then selection, and then merge sort. O(n log n) is less than O(n^2) so that is sort of to be expected, the merge sort being the fastest. Another metric to measure how efficient an algorithm is to use both best case and average case. Notably, selection sort’s best case is still Ω(n^2), whereas bubble sort and insertion sort both have a best case of Ω(n), and merge sort has a best case of Ω(n log n). This tells us that bubble sort and insertion sort can be fast, and even faster than merge sort or selection sort, but in the worst case, its slower. As for average cases, they’re all the same as the worst cases for the algorithms in this case.

## Conclusion

This project really emphasized to me how important algorithm efficiency really is. I already had prior experience with sorting and searching algorithms, but I never really saw how much the orders of growth really mattered.

## References

I used ChatGPT for the implementation of Binary search, selection sort, insertion sort, and bubble sort.

I used W3Schools for the implementation of merge sort.

## Appendix

### isSorted function

A computer screen with colorful text

Description automatically generated

### Linear Search function

A computer screen shot of a black screen with white text

Description automatically generated

### Binary Search function

A computer screen shot of a program

Description automatically generated

### Min search function

A screenshot of a computer

Description automatically generated

### Max search function

A screenshot of a computer

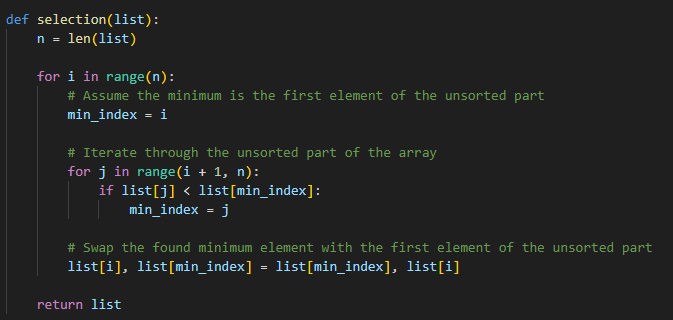
Description automatically generated

### Distinct search function

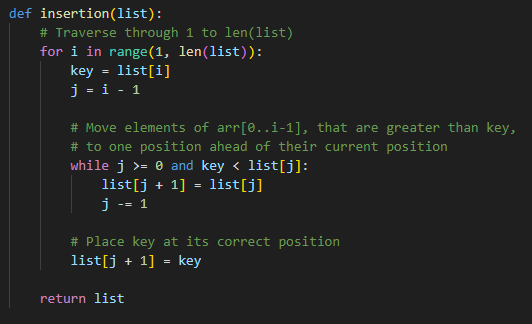
A screen shot of a computer program

Description automatically generated

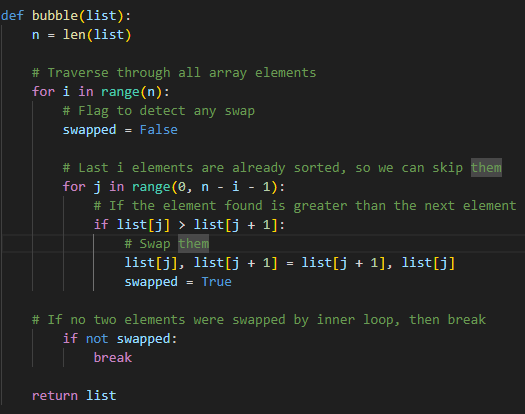
### Selection sort function



### Insertion sort function



### Bubble sort function



### Merge sort function

A screen shot of a computer program

Description automatically generated

### Test Driver

A screen shot of a computer program

Description automatically generated